Linear Models and Search
Part 1: Linear Regression

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

---

machine learning: the basic recipe

Abstract your problem to a **standard task.**
Classification, **Regression**, Clustering, Density estimation, Generative Modeling, Online learning, Reinforcement Learning, Structured Output Learning

Choose your **instances** and their **features**.
For supervised learning, choose a target.

Choose your **model class**.
Linear models

Search for a good model.
Choose a **loss function**, choose a **search method** to minimise the loss.

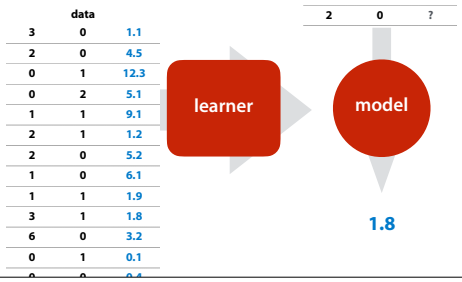
2

Here is the basic recipe for the last lecture. Today we'll have a look at linear models, both for regression and classification. We'll see how to define a linear model, how to formulate a loss function and how to search for a model that minimises that loss function.

Most of the lecture will be focused on search methods. The linear models themselves aren't that strong, but because they're pretty simple, we can use them to explain various search methods that we can also apply to more complex models

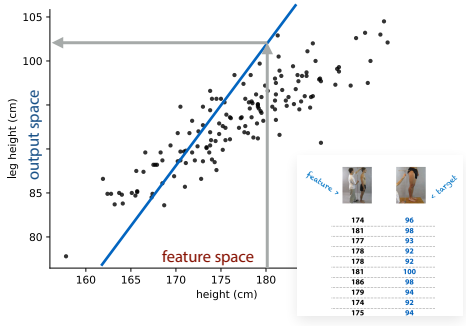
---

regression



3

We'll start with **regression**. Here's how we explained regression in the last lecture.

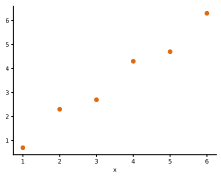


4

And here's the example we gave, of regression with a single feature (predicting the length of somebody's legs given their height).

regression: example data

x	t
1	0.7
2	2.3
3	2.7
4	4.3
5	4.7
6	6.3



We will start with **linear regression**, and use this dataset (with feature x and target variable y) as a running example.

We will assume that all our features are *numeric* for the time being. We will develop linear regression for an arbitrary number of features m, but we will keep visualizing what we're doing on this one-feature dataset.

---

notation

x, y, z scalar (i.e. a single number)

$\mathbf{x}, \mathbf{y}, \mathbf{z}$ vector (a column of numbers)

$\mathbf{X}, \mathbf{Y}, \mathbf{Z}$ matrix (a rectangular grid of numbers)

x_i scalar element of a vector **x**

X_{ij} scalar element of a matrix **X**

Throughout the course, we will use the following notation: lowercase non-bold for scalars, lowercase bold for vectors and uppercase bold for matrices.

When we're indexing *individual elements* of vectors and matrices, these are scalars, so they are non-bold.

---

multiple features

$$X = \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \ldots$$
$$T = t_1, t_2, t_3, \ldots$$

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix}$$

$\mathbf{x}_i$ Instance i in the data

x_j Feature j (of some instance)

X_{ij} Feature j of instance i

As we saw in the last lecture, an instance in machine learning is described by m *features* (with m fixed for a given dataset). We will represent this as a vector for each instance, with each element of the vector representing a feature.

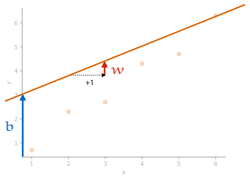
This can be a little confusing, since we sometimes want to index the instance of the dataset and sometimes the features of a given instance. Pay attention to whether the letter we're indexing is bold or non-bold.

We'll occasionally deviate from this notation when doing so makes things clearer, but we'll point it out when that happens.

---

defining a model: one feature

1 feature x_1: $f_{w,b}(\mathbf{x}) = wx_1 + b$



If we have one feature (as in this example) a standard linear regression model has two *parameters* (the numbers that determine which line we fit through our data): **w** the **weight** and **b,** the **bias**. The bias is also sometimes called the *intercept*.

b determines where the line crosses the vertical axis (or what value f takes when x = 0).

w determines how much the line rise if we move one step to the right (i.e. increase x by 1)

For the line drawn here, we have **b**=3 and **w** =0.5 (note that this isn't a very good fit for the data).

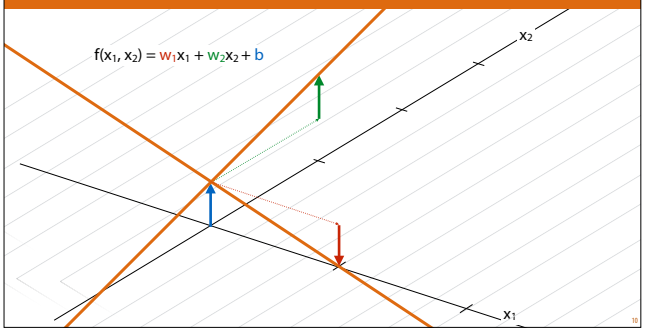
two features

1 feature x_1: $f_{w,b}(\mathbf{x}) = wx_1 + b$

2 features x_1, x_2: $f_{w_1,w_2,b}(\mathbf{x}) = w_1x_1 + w_2x_2 + b$

If we have two features, each feature gets its own **weight** (also known as a **coefficient**)

a 2D linear function



Here's what that looks like. The thick orange lines together indicate a plane (which rises in the x_2 direction, and declines in the x_1 direction). The parameter **b** describes how high above the origin this plane lies (what the value of f is if *both features* are 0). The value $\mathbf{w_1}$ indicates how much f increases if we take a step of 1 along the x_1 axis, and the value $\mathbf{w_2}$ indicates how much f increases if we take a step of size 1 along the x_2 axis.

for n features

$$f_{\mathbf{w},b}(\mathbf{x}) = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + b$$

$$= \mathbf{w}^T\mathbf{x} + b$$

$$\text{with } \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \text{ and } \begin{pmatrix} x_1 \\ \vdots \\ x_m \end{pmatrix}$$

For an arbitrary number of features, the pattern continues as you'd expect. We summarize the w's in a vector **w** with the same number of elements as **x**.

We call the w's the weights, and b the bias. The weights and the bias are the *parameters* of the model. We need to choose these to fit the model to our data.

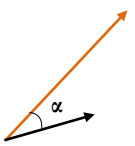
The operation of multiplying elements of **w** by the corresponding elements of **x** and summing them is the **dot product** of **w** and **x**.

dot product

$$\mathbf{w}^T\mathbf{x} \quad \text{or} \quad \mathbf{w} \cdot \mathbf{x}$$

$$\mathbf{w}^T\mathbf{x} = \sum_i w_ix_i$$

$$= ||\mathbf{w}||\ ||\mathbf{x}|| \cos\alpha$$

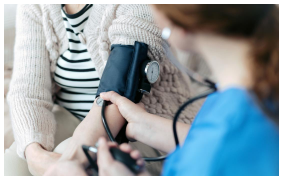


The **dot product** of two vectors is simply the sum of the products of their elements. If we place the features into a vector and the weights, then a linear function is simply their dot product (plus the b parameter).

The transpose (superscript T) notation arises from the fact that if we make one vector a row vector and one a column vector, and matrix-multiply them, the result is the dot product (try it).

The dot product also has a geometric interpretation: the dot product is equal to the lengths of the two vectors, multiplied by the cosine of the angle between them.

example: predicting high blood pressure

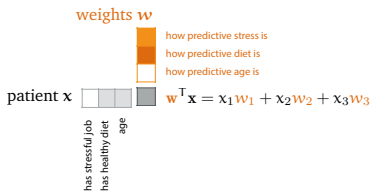


instances patients

features: job stress, healthy diet, age

To build some intuition on the meaning of **w**, let's look at an example. Imagine we are trying to predict the risk of high blood pressure based on these three features. We'll assume that the features are expressed in some number that measures these properties.

dot product



Here's what the dot product expresses. For some features, like job stress, we want to learn a positive weight (since more job stress should contribute to higher risk of high blood pressure). For others, we want to learn a negative weight (the healthier your diet, the lower your risk of high blood pressure). Finally, we can control the *magnitude* of the weights to control their relative importance: if age and job stress both contribute positively, but age is the bigger risk factor, we make both weights positive, but we make the weight for age bigger.

But which model fits our data best?

two more ingredients:
- loss function
- search method (next video)

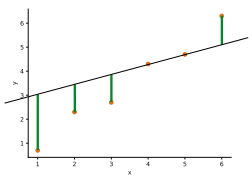
Given some data, which values should we choose for the parameter **w** and b?

In order to answer this question, we need two ingredients. First, we need a **loss function**, which tells us how well a particular choice of model does (for the given data) and second, we need a way to *search* the space of all models for a particular model that results in a low loss (a model for which the loss function returns a low value).

mean squared error loss

$$\text{loss}_{X,T}(p) = \frac{1}{n}\sum_j (f_p(x_j) - t_j)^2$$

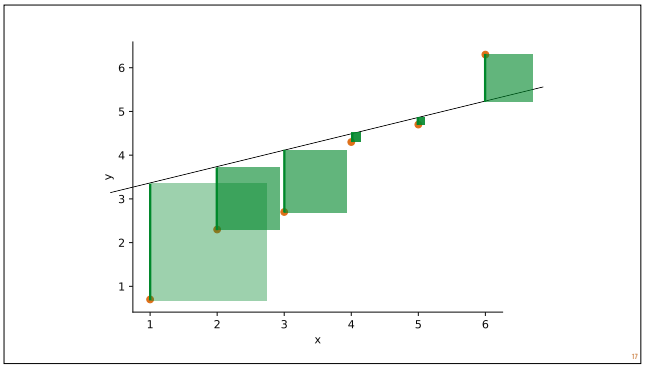
$$\text{loss}_{X,T}(w, b) = \frac{1}{n}\sum_j (w^T x_j + b - t_j)^2$$



Here is a common loss function for regression: the mean-squared error (MSE) loss. We saw this briefly already in the previous lecture.

Note that the loss function takes a *model* as its argument. The model maps the data to the output, the loss function maps a model to a loss value. The data functions as a constant in the loss function.

It takes the **residuals** (differences between the model predictions and actual data), squares them, and returns the average (or the sum, sometimes). The square, as noted before, is partly there to ensure that negative and positive residuals don't cancel out (giving us a small loss when we have big residuals).



But the squares also have another effect. They ensure that the big errors affect the loss more heavily than small errors. You can visualise this as shown here: the mean squared error is the mean of the areas of the green squares (it's also called *sum-of-squares loss*).

When we search for a well-fitting model, the search will try to reduce the big squares much more than the small squares.

If we think of the residuals as rubber bands, pulling on the regression line to pull it closer to the points, the rubber band on the bottom left pulls much harder than all the other ones. Therefore, any search algorithm trying to minimize this loss will be much more interested in moving the left of the line down than in moving the right of the line up.

Other loss functions exist, and this is just a simple one to start with. In a later lecture, we will see that this loss function follows from the assumption that model is correct except for added noise from a *normal distribution*.

Visualization stolen from **https://machinelearningflashcards.com/**

---

aside: slight variations

$$\sum_j (f_p(x^j) - y^i)^2$$

$$\frac{1}{n}\sum_j (f_p(x^j) - y^i)^2$$

$$\frac{1}{2}\sum_j (f_p(x^j) - y^i)^2$$

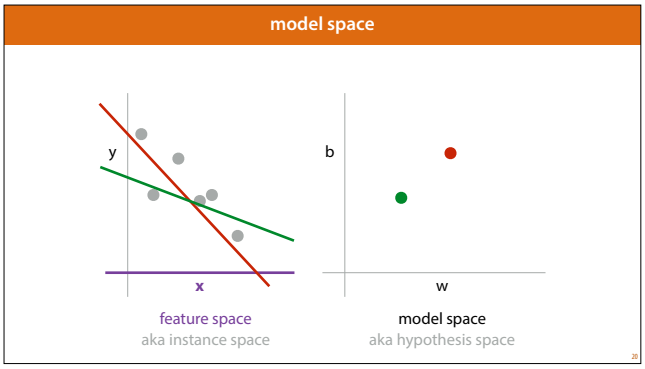
$$\sqrt{\frac{1}{n}\sum_j (f_p(x^j) - y^i)^2}$$

You may see slightly different versions of the MSE loss: sometimes we take the average, sometimes just the sum. Sometimes we multiply by 1/2 to make the derivative simpler. In practice, the differences don't mean much because we're not interested in the *absolute* value, just in how the loss changes from model to another.

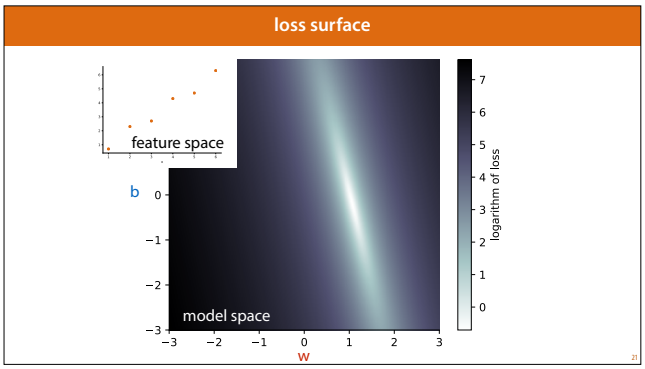
We will switch between these based on what is most useful in a given context.

Linear Models and Search
Part 2: Searching for a good model

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam



Remember the two most important spaces of machine learning: the instance space and the model space. The loss function maps every point in the model space to a loss value. Here, the **instance space** is just the x axis.

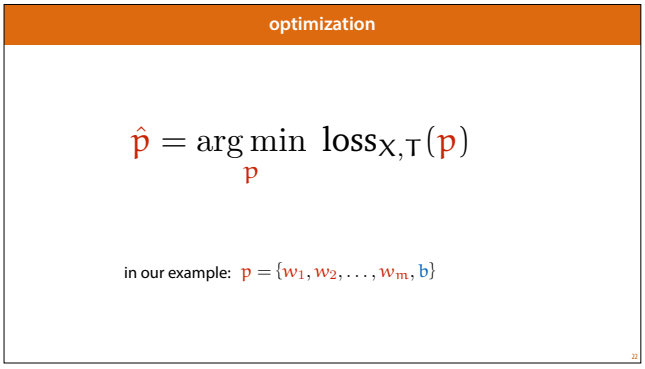


As we saw in the previous lecture, we can plot the loss for every point in our model space. This is called the **loss surface** or sometimes the **loss landscape**. If you imagine a 2D model space, you can think of the loss surface as a landscape of rolling hills, or sometimes of jagged cliffs.

Here is what that actually looks like for the two parameters of the one-feature linear regression. Note that this is specific to the data we saw earlier. For a different dataset, we get a different loss landscape.

To minimize the loss, we need to search this space to find the brightest point in this picture. Remember that, normally, we may have hundreds of parameters (one per feature) so it isn't as easy as it looks. Any method we come up with, needs to work in any number of dimensions.

We've plotted the logarithm of the loss as a trick to make this image visually easier to understand (it maps the values that are easy to tell apart to the values we care about). The logarithm is a monotonic function so log(loss(w, b)) has its minimum at the same place as loss(w, b).



The mathematical name for this sort of search is **optimization**. That is, we are trying to find the input (**p**, the model parameters) for which a particular function (the loss) is at its optimum (a maximum or minimum, in this case a minimum). Failing that, we'd like to find as low a value as possible.

We'll start by looking at some very simple approaches.

random search

start with a random point **p** in the model space

loop:

pick a random point **p′** close to **p**

if loss(**p′**) < loss(**p**):

p <- **p′**

Let's start with a very simple example: random search. We simply make random steps in the model space, and take a step back whenever the loss goes up.

You usually stop the loop when the loss gets to a pre-defined level, or you just run it for a fixed number of iterations.

---

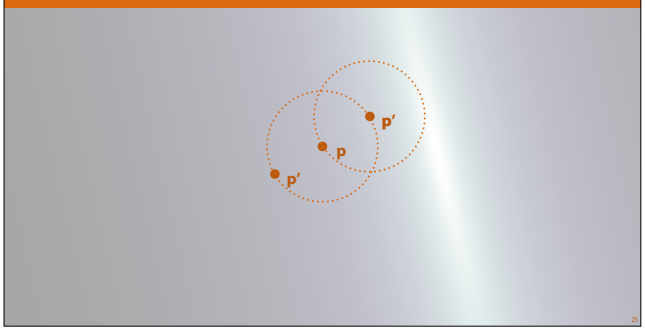
analogy: hiker in a snowstorm

A common analogy is a *hiker in a snowstorm*. Imagine you're hiking in the mountains, and you're caught in a snowstorm. You can't see a thing, and you'd like to get down to your hotel in the valley, or failing that, you'd like to get to as low a point as possible.

You take a couple of steps in every direction to see in which direction the mountain goes down quickest. You take a big step in that direction, and then repeat the process. This is, in effect, what random search is doing. More importantly, it's how blind random search is to the larger structure of the landscape.

image source: **https://www.wbur.org/hereandnow/2016/12/19/rescue-algonquin-mountain**

"close to"

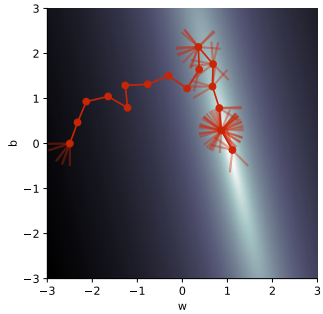


To implement the random search we need to define how to pick a point "close to" another in model space.

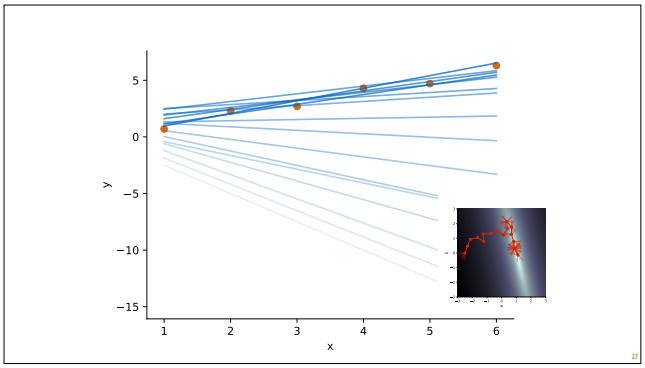
One option is to choose the next point by sampling uniformly among all points with some pre-chosen distance **r** from w. Formally: it picks from the hypersphere (or circle, in 2D) with radius **r**, centered on **p**.

---

random search

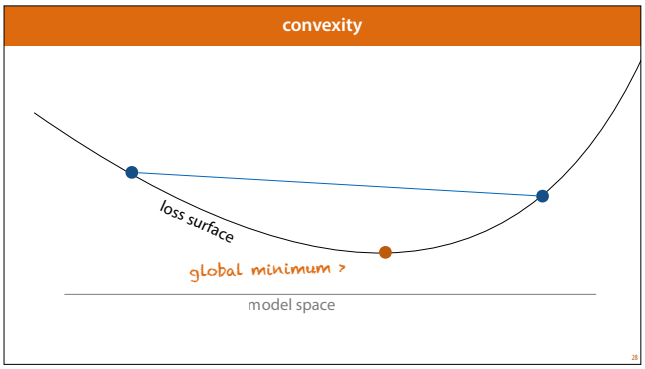


Here is random search in action. The transparent red offshoots are successors that turned out to be worse than the current point. The algorithm starts on the left, and slowly (with a bit of a detour) stumbles in the direction of the low loss region.



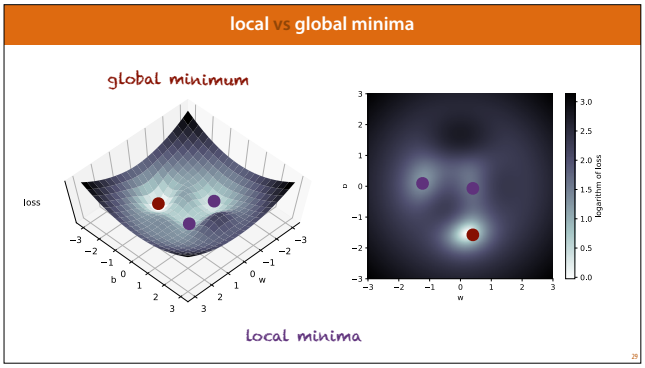
Here is what it looks like in **feature space**. The first model (bottom-most line) is entirely wrong, and the search slowly moves, step by step, towards a reasonable fit on the data.

Every blue line in this image corresponds to a red dot in the model space (inset).



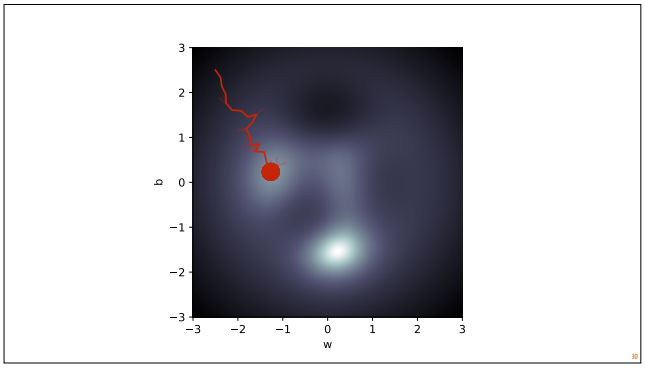
One of the reasons such a simple approach works well enough for our problem is that our problem is **convex**. A surface (like our loss landscape) is convex if a line drawn between any two points on the surface lies entirely above the surface. One of the implications of convexity is that any point that looks like a minimum locally (because all nearby points are higher) it must be the **global minimum**: it's lower than any other point on the surface.

This minimum is the optimal model. So long as we know we're moving down (to a point with lower loss), we can be sure we're moving towards the minimum.



So let's look at what happens if the loss surface isn't convex: what if the loss surface has multiple **local minima**?

Here's a loss surface (not based on an actual model with data, just some function) with a more complex structure. The two purple points are the lowest point in their respective *neighborhoods*, but the red point is the lowest point globally.



Here we see random search on our more complex loss surface. As you can see, it makes a beeline for one of the local minima, and then gets stuck there. No matter how many more iterations we give it, it will never escape.

Note that changing the step size will not help us here. Once the search is stuck, it stays stuck.

simulated annealing

pick a random point **p** in the model space

loop:

  pick a random point **p′** close to **p**

  if loss(**p′**) < loss(**p**):

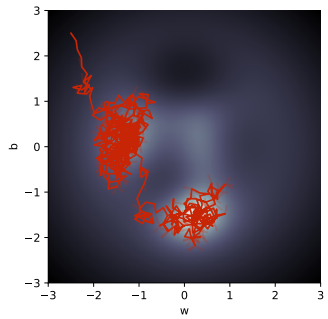
    p <- **p′**

  else:

    with probability q: **p** <- **p′**

31

Here's a simple trick that can help us escape local minima: if the next point chosen isn't better than the current one, we still pick it, but only with some small probability. In other words, we allow the algorithm to *occasionally* travel uphill. This means that whenever it gets stuck in a local minimum, it still has some probability of escaping, and finding the global minimum.



32

Here is a run of simulated annealing. Of course, with SA there is always the possibility that it will jump out of the global minimum again and move to a worse minimum. That shouldn't worry us, however, so long as we remember the best model we've observed. Then we can just let SA jump around the model space driven partly by random noise, and partly by the loss surface.

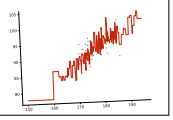
---

machine learning vs. optimization

optimization: find the minimum, or the best possible approximation

machine learning: find the lowest loss that *generalizes*

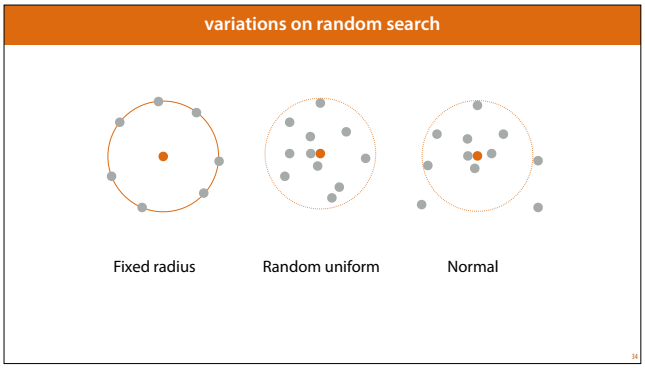
Minimize the loss on the test data, seeing only the training data.



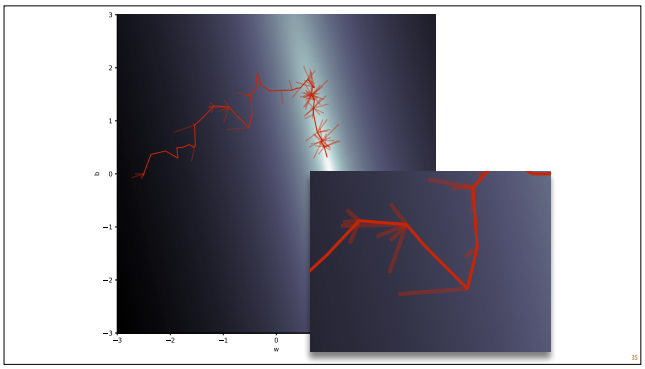
We often frame machine learning as an optimization problem, and we use many techniques from optimization, but it's important to recognize that there is a difference.

Optimization is concerned with finding the absolute minimum (or maximum) of a function. The lower the better, with no ifs or buts. In machine learning, if we have a very expressive model class (like the regression tree from the last lecture), the model that actually minimizes the loss on the training data is the one that overfits. In such cases, we're not looking to minize the loss on the training data, since that would mean overfitting, we're looking to minimize the loss on the *test data*. Of course, we don't get to see the test data, so we use the training data as a stand, and try to control against overfitting as best we can.

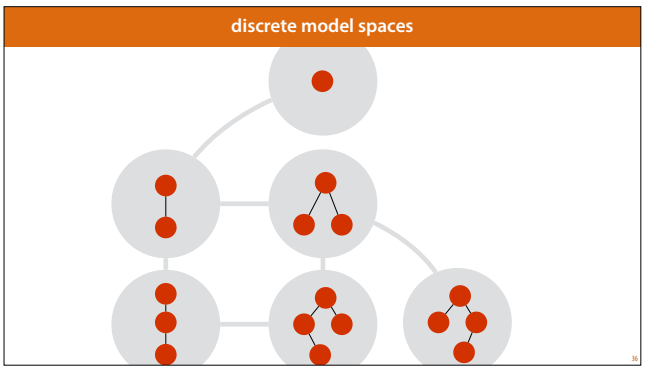
In the case of underpowered models like the linear model, this distinction isn't too important, since they're very unlikely to overfit. Here, the model that minimizes the loss on the training data is likely the model that minimizes the loss on the test data as well.



The fixed step size we used so far is just one way to sample the next point. To allow the algorithm to occasionally make smaller steps, you can sample m' so that it is *at most* some distance away from m, instead of *exactly*. Another approach is to sample the distance from **a Normal distribution**. That way, most points will be close to the original m, but every point in the model space can theoretically be reached in one step.



Here is what random search looks when the steps are sampled from a normal distribution. Note that the "failed" steps all have different sizes.

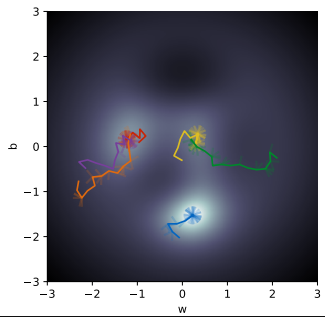


The space of linear models is *continuous*: between every two models, there is always another model, no matter how close they are together. * If your model space is discrete, for instance in the case of tree models, you can still apply random search and simulated annealing. You just need to define which models are "close" to each other. Here we say that two trees are close if I can turn one into the other by adding or removing a single node.

Random search and SA can now be used to search this graph to find the tree model that gives the best performance. Note that in practice, we usually use a different method to search for decision trees and regression trees. We will introduce this algorithm in a later lecture.

* Strictly speaking this is not a correct definition of a continuous space. It suffices for our purposes.

parallel search



Another thing you can do is just to run random search a couple of times independently (one after the other, or in parallel). If you're lucky one of these runs may start you off close enough to the global minimum.

For simulated annealing, doing multiple runs makes less sense. There's not much difference between 10 runs of 100 iterations and one run of 1000. The only reason to do multiple runs of SA is because it's easier to parallelize over multiple cores or machines.

---

population methods

evolutionary algorithms

- genetic algorithms
- evolutionary strategies

particle swarm optimization

ant colony optimization

To make parallel search even more useful, we can introduce some form of communication between the searches happening in parallel. If we see the parallel searches as a population of agents that occasionally "communicate", we can guide the search a lot more. Here are some examples. we won't go into this too deeply. We will only take a (very) brief look at evolutionary algorithms.

Often, there are specific variants for discrete and for continuous model spaces.

---

evolutionary algorithms

Start with a *population* of k models.

loop:

rank the population by loss

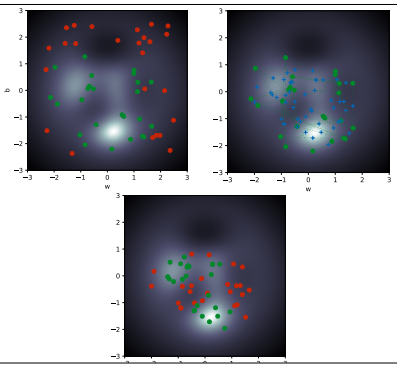
remove the half with the worst loss

"breed" a new population of k models

optional: add a little noise to each child.

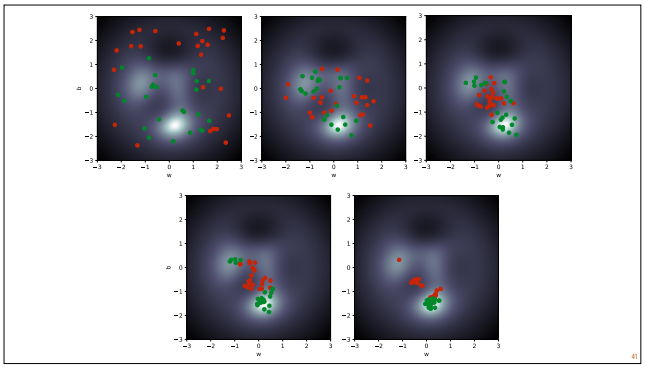
Here is a basic outline of an evolutionary method (although many variations exist). In order to instantiate this, we need to define what it means to "breed" a population of new models from an existing population. A common approach is to select to random parents and to somehow average their models. This is easy to do in a continuous model space (we can literally average the two parent models to create a child). In a discrete model space, it's more difficult, and it depends on the specifics of the model space.

The breeding process (sometimes called the **crossover operator**) is usually the most difficult part of designing an evolutionary algorithm.



Here's what that looks like. We start with a population of 50 models, and compute the loss for each. We kill the worst 50% (the red dots) and keep the best 50% (the green dots). We then create a new population (the blue crosses), by randomly pairing up parents from the green population, and taking the point halfway between the two parents, with a little noise added.

We then take the blue crosses as the new population and iterate.



Here are five iterations of the algorithm. Note that in the intermediate stages, the population covers both the local and the global minima.

population methods

Powerful

Easy to parallelise

Slow/expensive for complex models

Difficult to tune

To make parallel search even more useful, we can introduce some form of communication between the searches happening in parallel. If we see the parallel searches as a population of agents that occasionally "communicate", we can guide the search a lot more. Here are some examples. we won't go into this too deeply. We will only take a (very) brief look at evolutionary methods.

Often, there are specific variants for discrete and for continuous model spaces.

To escape local minima:

- add randomness, add multiple models

To converge faster:

- combine known good models (population methods)

black box optimization

random search, simulated annealing:

- very simple
- we only need to compute the loss function for each model
- can require many iterations
- also works for discrete model spaces (like tree models)

All these search methods are instances of **black box optimization**.

Black box optimization refers to those methods that only require us to be able to compute the loss function. We don't need to know anything about the internals of the model.

In the next video we'll look at a way to improve the search by opening up the black box: gradient descent.

Linear Models and Search
Part 3: Gradient Descent

Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

---

towards gradient descent: branching search

pick a random point **p** in the model space

loop:

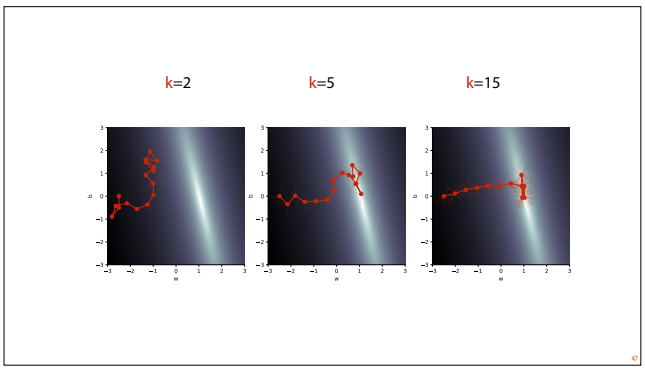
pick **k** random points {p_i} close to **p**

p' <- argmin$_{p_i}$ loss(p_i)

if loss(**p'**) < loss(**p**):

p <- **p'**

As a stepping stone to what we'll discuss in this video, let's take the random search from the previous lecture, and add a little more inspection of the local neigborhood before taking a step. Instead of taking one random step, we'll look at k random steps and move in the direction of the one that gives us the lowest loss.

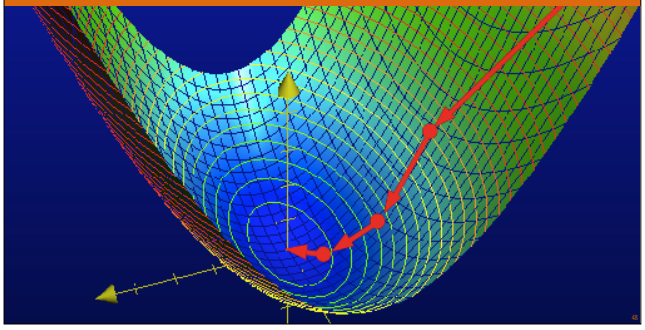


As you can see, the more samples we take, the more directly we head for the region of low loss. The more closely we inspect our local neighbourhood, to determine in which direction the function decreases quickest, the faster we converge.

The lesson here is that the better we know in which direction the loss decreases, the faster our search converges. In this case we pay a steep price, we have to evaluate our function 15 times to work out a better direction.

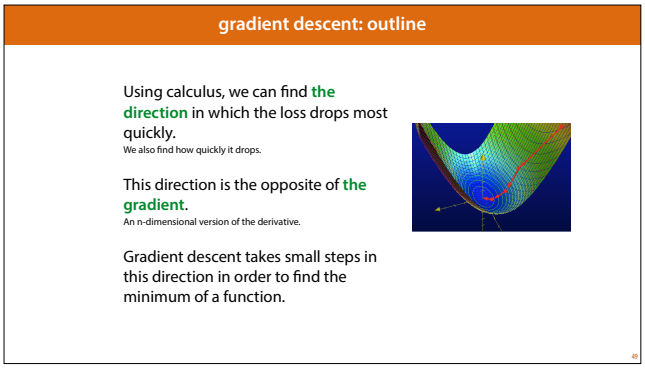
---

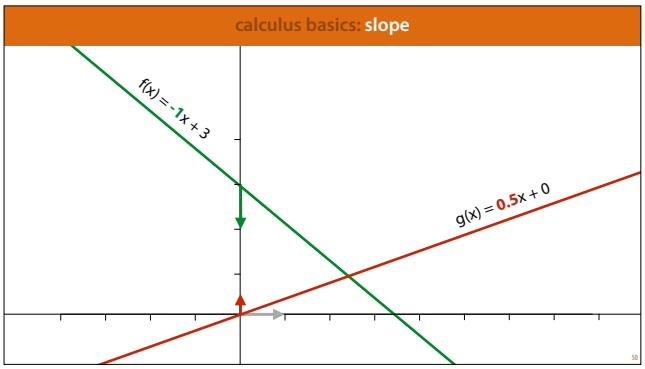
gradient descent



However, if our model space is continuous, and if our loss function is smooth, we don't *need* to take multiple samples to guess the direction of fastest descent: *we can simply derive it, using calculus*. This is the basis of the **gradient descent algorithm**.

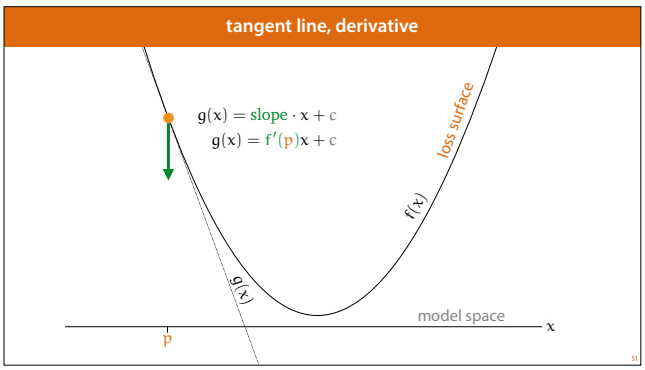
image source: **http://charlesfranzen.com/posts/multiple-regression-in-python-gradient-descent/**





Before we dig in to the gradient descent algorithm, let's review some basic principles from calculus. First up, **slope**. The slope of a linear function is simply **how much it moves up** if **we move one step to the right**. In the case of f(x) in this picture, the slope is *negative*, because the line moves down.

In our 1D regression model, the parameter w was the slope.



The **tangent line** of a function at particular point x is the line that just touches the function at x. The **derivative of the function gives us the slope of the tangent line**. Traditionally we find the minimum of a function by setting the derivative equal to 0 and solving for x. This gives us the point where the tangent line has slope 0, and is therefore horizontal.

For complex models, it may not be possible to solve for x in this way. However, we can still use the gradient to *search* for the minimum. Looking at the example above, we note that the tangent line moves down (i.e. the slope is negative). This tells us that we should move to the right to follow the function downward. As we take small steps to the right, the derivative stays negative, but gets smaller and smaller as we close in on the minimum. This suggests that the *magnitude* of the slope lets us know how big the steps are that we should take, and the *sign* gives us the direction.

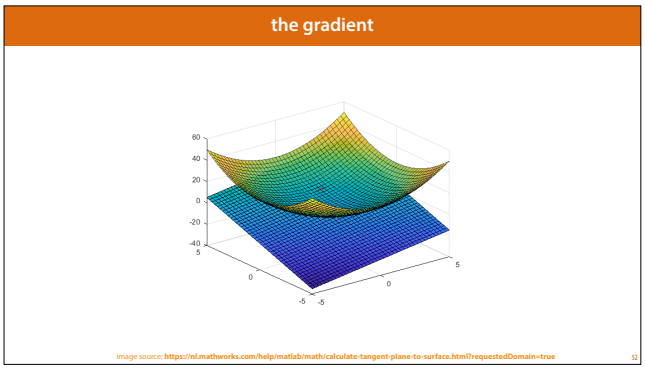
A useful analogy is to think of putting a marble at some point on the curve and following it as it rolls downhill to find the lowest point.

What we need to do now, is to take the derivative of the function describing the loss surface, and work out

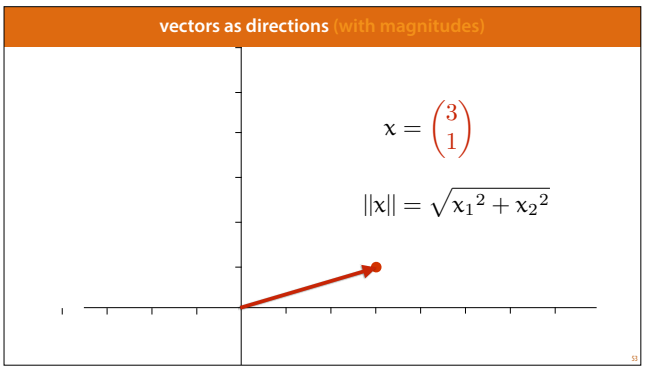
a) The direction in which the function decreases the quickest. We call this **the direction of steepest descent**.

b) How quickly the function decreases in that

direction.

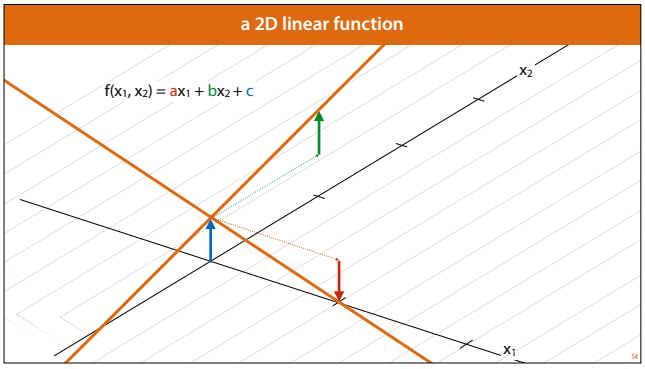


To apply this principle to machine learning, we'll need to generalise it for loss functions with multiple inputs (i.e. models with multiple parameters). We do this by generalising the *derivative* to the ***gradient***. The tangent line becomes a tangent (hyper)plane. The hyperplane gives us both a direction to move in, and an indication of how big a step we should take in that direction.



We're used to thinking of vectors as points in the plane. But they can also be used to represent *directions*. In this case we use the vector to represent the arrow from the origin to the point. This gives us a **direction** (the direction in which the arrow points), and a **magnitude** (the length of the arrow).

So this direction of steepest descent that we're looking for (in model space) can be expressed as a vector.

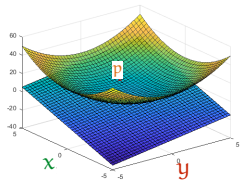


Remember, that this is how we express a linear function in n dimensions: we assign each dimension a slope, and add a single bias (c).

In this image, the two weights of a linear 2D function (a and b) representing a hyperplane, are just one slope per dimension. If we move one step in the direction of x_1, we move up by a, and if we move one step in the direction of x_2, we move up by b.

gradient

$$\nabla f(x, y) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$$



tangent hyperplane:

$$g(x, y) = f'_x(p)x + f'_y(p)y + c$$

$$g(\mathbf{x}) = \nabla f(p)^T \mathbf{x} + c$$

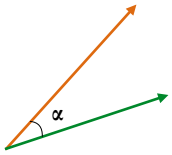
We are now ready to define the gradient. Any function from n inputs to one output has n variables for which we can take the derivative. These are called partial derivatives: they work the same way as regular derivatives, except that you when you take the derivative with respect to one variable x, you treat the other variables as constants. For technical reasons, the derivative is *row* vector, not a column vector.

If a particular function f(x) has gradient **g** for input x, then f'(**x**) = **g**^{T}**x** + b (for some b) is the tangent hyperplane at **x**.

---

the direction of steepest ascent

$$\begin{aligned} g(\mathbf{x}) &= \mathbf{w}^T\mathbf{x} + b \\ &= ||\mathbf{w}||\ ||\mathbf{x}|| \cos\alpha \\ ||\mathbf{x}|| &= 1 \\ &\rightarrow ||\mathbf{w}|| \cos\alpha \end{aligned}$$



So, now that we have a local linear approximation to our function, which is the direction of steepest ascent on that approximation?

Since g is linear, many details don't matter: we can set b to zero, since that just translates the hyperplane up or down. It doesn't matter *how big* a step we take in any direction, so we'll take a step of size 1. Finally, it doesn't matter where we start from, so we will just start from the origin. So the question becomes: for which input of magnitude 1 does g provide the biggest output?

To see the answer, we need to use the geometric interpretation of the dot product. Since we required that ||**x**||= 1**,** this disappears from the equation, and we only need to maximise the quantity ||**w**|| cos(α) (where only α depends on our choice of input, **w** is given). cos(α) is maximal when α is zero: that is, when **x** and **w** are pointing in the same direction.

In short: w, the gradient, *is* the direction of steepest ascent. This means that -**w** is the direction of steepest *descent*.

---

summary

The tangent hyperplane of a function f *approximates* the function f locally.

The gradient of f gives the *slope* of the tangent hyperplane.

The vector expressing the slope is also the *direction of steepest ascent.*

The opposite vector is the direction of steepest descent.

gradient descent

pick a random point **p** in the model space

loop:

p ← **p** - η ∇loss(**p**)

we usually set η somewhere between 0.0001 and 0.1

58

Here is the **gradient descent algorithm**. Starting from some candidate **p**, we simply compute the gradient at **p**, subtract it from the current choice, and iterate this process:

- We *subtract*, because the gradient points *uphill*. Since the gradient is the direction of steepest ascent, the negative gradient is the direction of steepest *de*scent.
- Since the gradient is only a *linear approximation* to our loss function, the bigger our step the bigger the approximation error. Usually we scale down the step size indicated by the gradient by multiplying it by a learning rate η. This value is chosen by trial and error, and remains constant throughout the search.

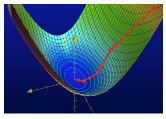
Note a potential point of confusion: we have two linear functions here. One is the *model*, whose parameters are indicated by **w** and b. The other is the tangent hyperplane to the loss function, whose slope is indicated by ∇loss(**p**) here. These are different functions on different spaces.

We can iterate for a fixed number of iterations, until the loss gets low enough, or until the gradient gets close enough to the zero vector.

---

gradient descent: outline

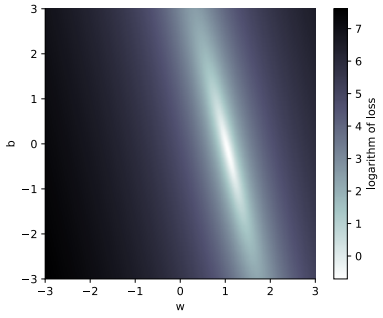
Using calculus, we can find **the direction** in which the loss drops most quickly.
We also find how quickly it drops.



This direction is the opposite of **the gradient**.
It is an n-dimensional version of the derivative.

Gradient descent takes small steps in this direction in order to find the minimum of a function.
Like a marble rolling down a hill

59



60

Let's go back to our example problem, and see how we can apply gradient descent here.

Unlike random search, it's not enough to just compute the loss for a given model, **we need the gradient of the loss**.

$$\text{loss}(w, b) = \frac{1}{n}\sum_i (wx_i + b - t_i)^2$$

$$\nabla\text{loss}(w, b) = \left(\frac{\partial\text{loss}(w, b)}{\partial w}, \frac{\partial\text{loss}(w, b)}{\partial b}\right)$$

Here is our loss function again, and the two partial derivatives we need work out to find the gradient.

To simplify the notation we'll let x_i refer to the only feature of instance i.

$$\begin{aligned}
\frac{\partial\text{loss}(w, b)}{\partial w} &= \frac{\partial \frac{1}{n}\sum_i (wx_i + b - t_i)^2}{\partial w} \\
&= \frac{1}{n}\sum_i \frac{\partial (wx_i + b - t_i)^2}{\partial w} \\
&= \frac{1}{n}\sum_i \frac{\partial (wx_i + b - t_i)^2}{\partial (wx_i + b - t_i)} \frac{\partial (wx_i + b - y_i)}{\partial w} \\
&= \frac{2}{n}\sum_i (wx_i + b - t_i)x_i \\
\frac{\partial\text{loss}(w, b)}{\partial b} &= \frac{\partial \frac{1}{n}\sum_i (wx_i + b - t_i)^2}{\partial b} \\
&= \frac{2}{n}\sum_i (wx_i + b - t_i)
\end{aligned}$$

Here are the derivations of the two partial derivatives:

- first we use the *sum rule*, moving the derivative inside the sum symbol
- then we use the *chain rule*, to split the function into the composition of computing the residual and squaring, computing the derivative of each with respect to its argument.

The second homework exercise provides a list of the most common rules for derivatives.

---

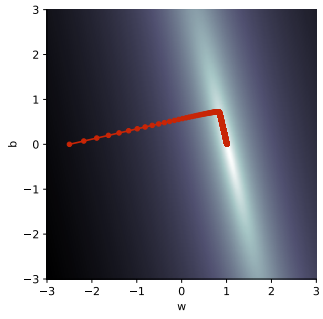
gradient descent for our example

pick a random point (w, b) in the model space

loop:

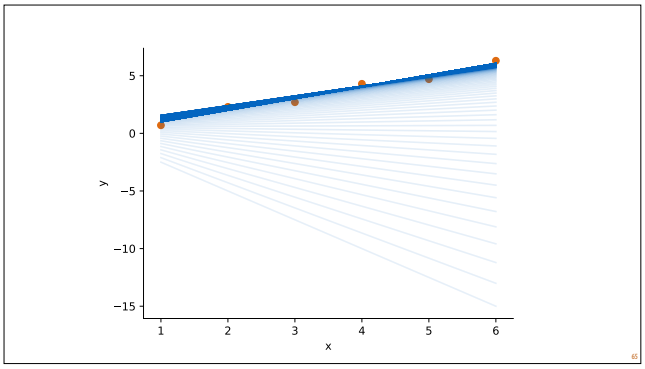
$$\begin{pmatrix} w \\ b \end{pmatrix} \leftarrow \begin{pmatrix} w \\ b \end{pmatrix} - \eta \begin{pmatrix} \frac{2}{n}\sum_i (wx_i + b - t_i)x_i \\ \frac{2}{n}\sum_i (wx_i + b - t_i) \end{pmatrix}$$

next guess *current guess* gradient

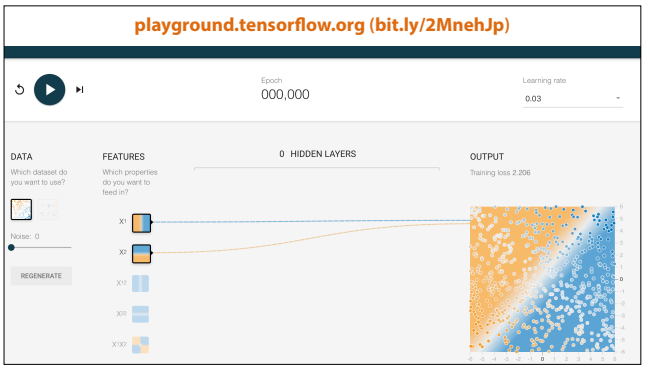


Here is the result. Note how the iteration converges directly to the minimum. Note also that we have no rejections: the algorithm is fully deterministic: it computes the optimal step, and takes it.

Finally, note that while we have focused mainly on the direction that the gradient gives us, it actually also controls the **step size**. As we get closer to the minimum, the function *flattens out* and the magnitude of the gradient decreases. The effect is that as we approach the minimum the algorithm takes smaller and smaller steps, preventing us overshooting the optimum.



Here is what it looks like in feature space.

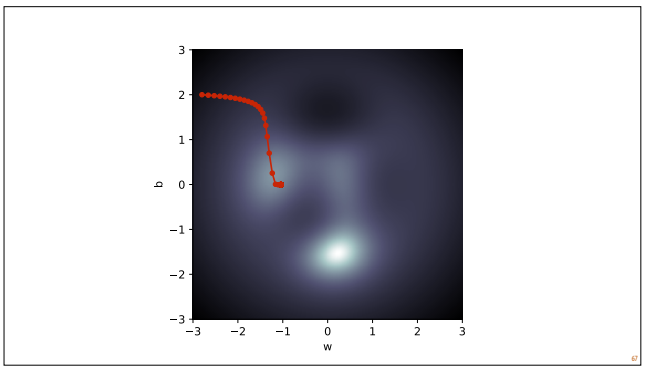


https://bit.ly/36r66Qh

Here is a very helpful little browser app that we'll return to a few times during the course. If you use this shortened link, you'll see the stripped down version containing only things we've discussed so far.

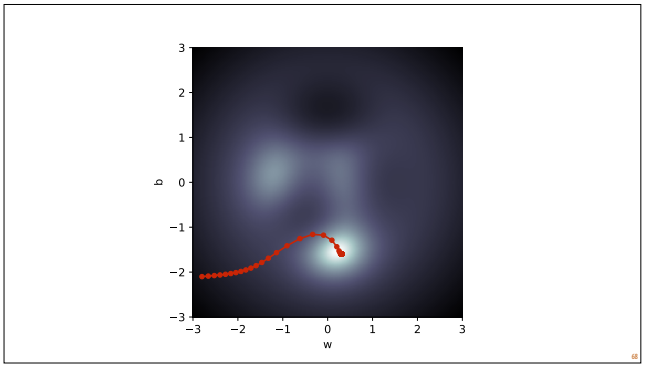
The output for the data is indicated by the color of the points, the output of the model is indicated by the colouring of the plane.

Note that the page calls this model a neural network (which we won't discuss for a few more weeks). Linear models are just a very simple neural network.



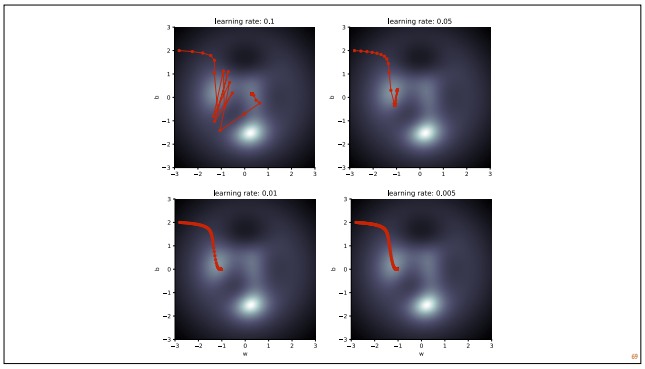
If our function is non-convex, gradient descent doesn't help us with local minima. As we see here, it heads straight for the nearest minimum and stays there. To make the algorithm more robust against this type of thing, we need to add a little randomness back in, preferably without destroying the behaviour of moving so cleanly to a minimum once one is found.

We can try multiple starting points. Later we will see *stochastic* gradient descent, which computes the gradient only over subsets of the data (making the algorithm more efficient, and adding a little randomness at the same time).



Here is a more fortunate run.

NB: The point of convergence seems a little off in these images. The partial derivatives for this function are very complex (I used **Wolfram Alpha** to find them), so most likely, the implementation has some numerical instability.



Here, we see the effect of the learning rate. If we set if too high, the gradient descent jumps outof the first minimum it finds. A little lower and it stays in the neighborhood of the first minimum, but it sort of bounces from side to side, only very slowly moving towards the actual minimum.

At 0.01, we find a sweet spot where it finds the local minimum pretty quickly. At 0.005 we see the same behavior, but we need to wait much longer, because the step sizes are so small.

---

gradient descent

only works for continuous model spaces

… with smooth loss functions

… for which we can work out the gradient

does not escape local minima

very fast, low memory

very accurate

backbone of 99% of modern machine learning.

---

but actually…

$$\frac{\partial \text{loss}(w, b)}{\partial w} = 0$$
$$\frac{\partial \text{loss}(w, b)}{\partial b} = 0$$

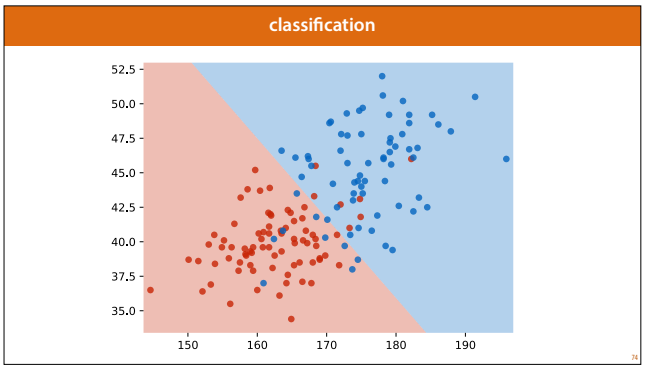
there's an analytical solution (for this model)

It's worth saying that for **linear regression**, although it makes a nice, simple illustration, none of this is actually necessary. For linear regression, we can set the derivatives equal to zero and solve explicitly for **w** and for **b**. This would give us the optimal solution without searching.

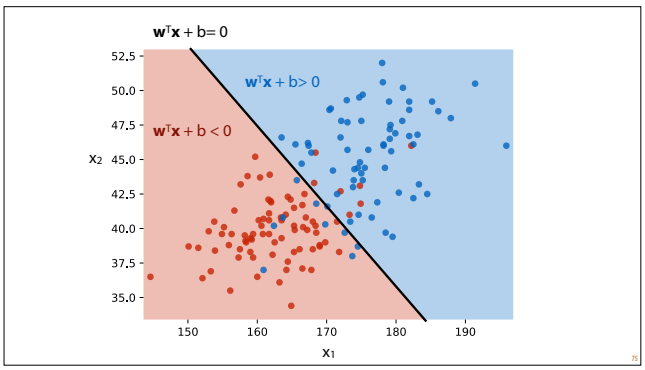
However, this trick will stop working very quickly once we start looking at more complicated models, so we won't go down this route very much.

Linear Models and Search
Part 4: Gradient Descent and Classification

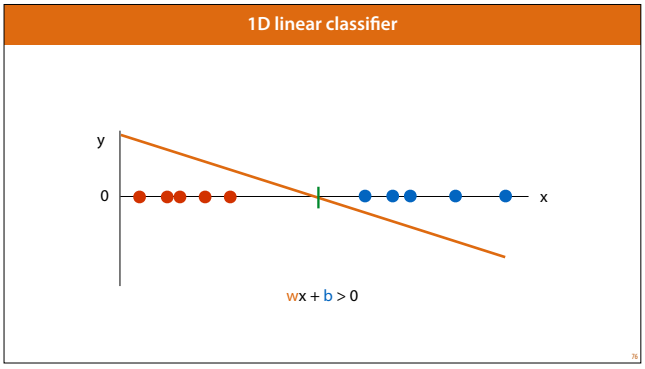
Machine Learning
mlvu.github.io
Vrije Universiteit Amsterdam

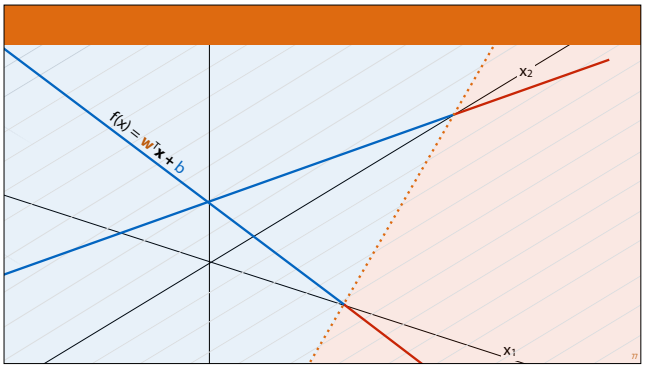


Now, let's look at how this works for classification. How do we define a linear classifier: that is a classifier whose decision boundary is always a line in instance space.

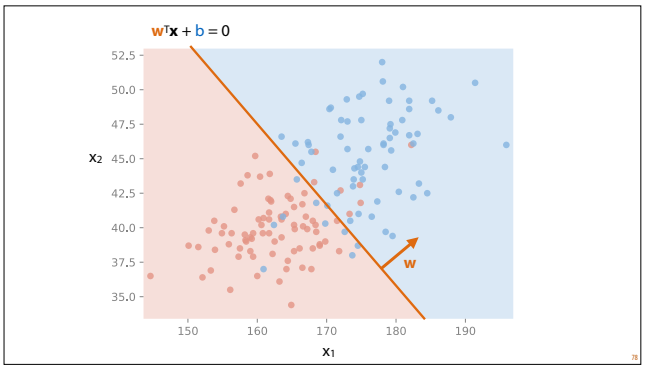


To define a linear decision boundary, we take the same functional form: some weight vector **w**, and a bias b. If $\mathbf{w}^T\mathbf{x} + b$ is larger than 0, we call **x** one class, if it is smaller than 0, we call it the other (we'll stick to binary classification for now).





The function f(x) = $\mathbf{w}^T\mathbf{x}$ + b describes a linear function from our feature space to a single value. Here it is in 2D: a plane that intersects the feature space. The line of intersection is our decision boundary.

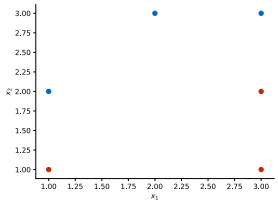


This also shows us how to interpret **w**. Since it is the direction of steepest ascent, it is the vector **perpendicular to the decision boundary**, pointing to the class we assigned to the case where $\mathbf{w}^T\mathbf{x}$ + b is larger than 0 (the blue class in this case).

---

example data

x_1	x_2	
1	2	true
1	1	false
2	3	true
3	3	true
3	1	false
3	2	false



Here is a simple classification dataset, which we'll use to illustrate the principle.

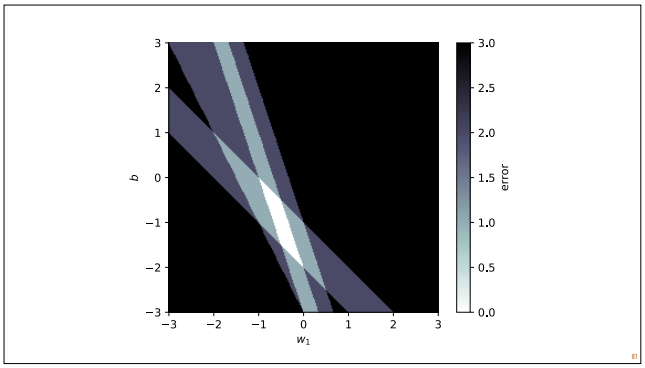
---

loss?

nr. of misclassified examples (error)?

This gives us a model space, but how do we decide the quality of any particular model? What is our **loss function** for classification?

The thing we are (usually) trying to minimise is the **error**: the number of misclassified examples.



This is what our loss surface looks like for that loss function. Note that it consists almost entirely of flat regions. This is because changing a model a tiny bit will usually not change the number of misclassified examples. And if it does, the loss function will suddenly jump a lot.

In these regions, random search would have to do a random walk, stumbling around until it finds a ridge by accident.

Gradient descent would fare even worse: the gradient is zero everywhere in this picture, except exactly on the ridges, where it is undefined. Gradient descent would either crash, or simply never move.

Note that our model now has three parameters w_1, w_2 and b. In order to plot the loss surface in two dimensions, we have fixed w_2=1.

Sometimes your loss function

should not be the same as

your evaluation function.

This is an important lesson about loss functions. They serve two purposes:

1. to express what quality we want to maximise in our search for a good model
2. to provide a smooth loss surface, so that we can find a path from a bad model to a good one

For this reason, it's common not to use the error as a loss function, even though it's the thing we're actually interested in minimizing. Instead, we'll replace it by a loss function that has its minimum at (roughly) the same model, but that provides a smooth, differentiable loss surface.

---

classification losses

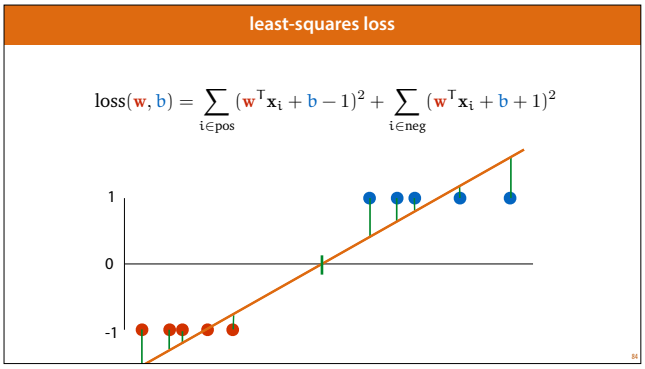
Least squares loss (this video)

Log loss / Cross entropy (Lecture 5, Probability)

SVM loss (Lecture 6, Linear Models 2)

In this course, we will investigate three common loss functions for classification. The first, least-squares loss, is just an application of MSE loss to classification, we will discuss that in the remainder of the lecture.

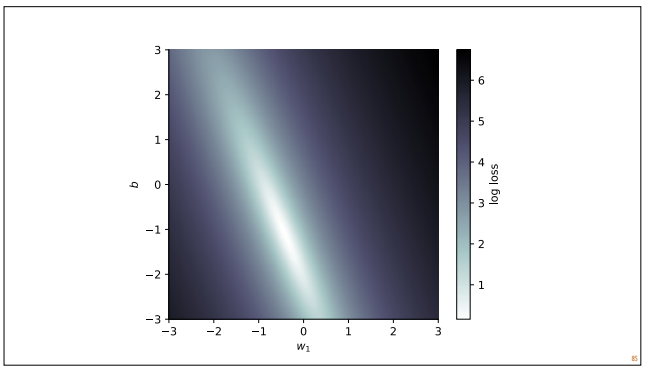
The others require a bit more background, so we'll save them for later.



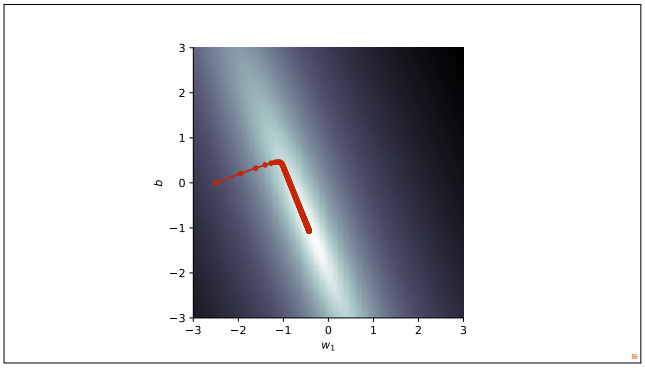
There are a few loss functions available for classification. We'll give you a simple one now, the least squares loss for classification, and come back to this problem later.

The least squares classifier essentially turns the classification into a regression problem: it assigns positive points the numeric value +1 and negative points the value -1, we then use a basic MSE loss that we saw before the break. Since we are look for a linear function that is positive for point in the positive class and negative for points in the negative class, this is a reasonable approach.

Performing gradient descent with this loss function will result in a line that minimised the green residuals. Hopefully the points are far apart that the decision boundary(the **single point** where the orange line crosses the x axis) separates the two classes.

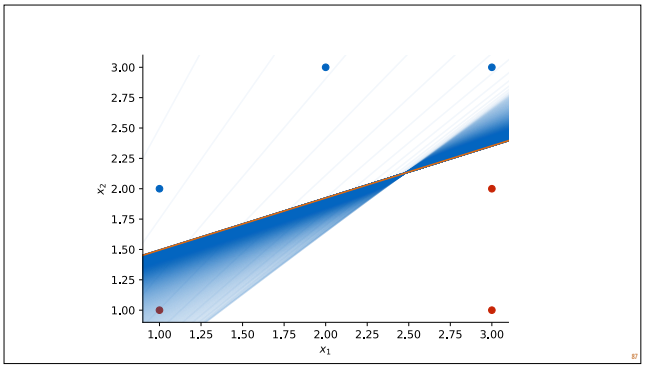


With this loss function, we note that our loss surface is perfectly smooth.

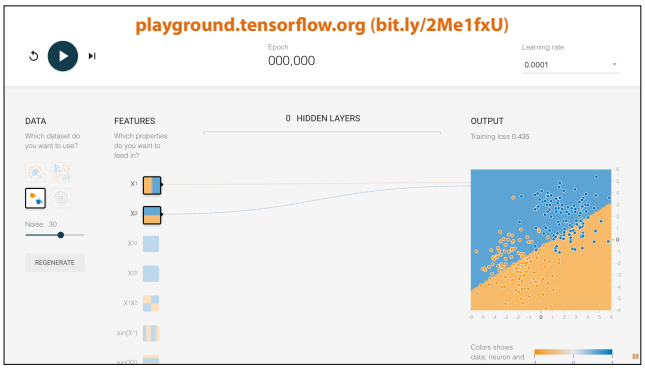


And gradient descent has no problem finding a solution.

Note, however that the optimum under this loss function may not perfectly separate the classes, even if they *are* linearly separable. We'll see some other loss functions later that provide a better optimum as well as a smooth loss surface.



Here is the result in instance space, with the final decision boundary in orange.



https://bit.ly/2Me1fxU

The tensorflow playground also allows us to play around with linear classifiers. Note that only for one of the two datasets, the linear decision boundary is appropriate.

This example actually uses a logistic regression loss, rather than a least squares loss. We'll discuss logistic regression (which, confusingly, is a classification method) in the first probability lecture.

---

summary

Black box optimization:
Random search, Simulated Annealing, Evolutionary
Simple, works on discrete model spaces

Gradient descent:
Powerful, only on continuous model spaces
Very important, will see again

For classification:
Find a smooth loss function
least squares loss (more on this later)

---

mlcourse@peterbloem.nl